C12

DDRSTsvc_init   5   (EDMDDcr_rstsvc.cc)
EDMDDSvcInit.............14   (EDMDDcr_rstsvc.cc)
LockSvcMutex   2   (EDMDDcr_rstsvc.cc)
UnlockSvcMutex...........4   (EDMDDcr_rstsvc.cc)
edmrst_create_ddp_client_connection   12   (EDMDDcr_rstsvc.cc)
edmrst_send_chndl_to_private_svc....10   (EDMDDcr_rstsvc.cc)
edmrst_send_uid_to_private_svc   11   (EDMDDcr_rstsvc.cc)

```
1   /*
2   ** =========================================================================
3   **
4   ** Copyright 1996,1997 EMC Corporation
5   **
6   */
7   /*
8   ** =========================================================================
9   **
10  ** DDRSTsvc_init.c
11  **
12  ** =========================================================================
13  **
14  ** Mission Statement:
15  **
16  ** =========================================================================
17  **
18  ** Primary Data Acted On:
19  **
20  ** =========================================================================
21  **
22  ** Compile-Time Options:
23  **
24  **     USE_SUNRPC  - Compile source with sunrpc support.   If
25  **                   not set, assume DCE support.
26  **
27  ** =========================================================================
28  **
29  ** =========================================================================
30  */
31  #if !defined(lint)
32  static char     RCS_id [] = "@(#)$RCSfile: EDMccr.c,v $ "
33                  "$Revision: 1.23 $ "
34                  "$Date: 1997/02/06 20:49:15 $" ;
35  #endif

37  /* #define _POSIX_SOURCE       unable to compile with this define set */
38  /* #define _XOPEN_SOURCE       unable to compile with this define set */

40  #include <sys/types.h>
41  #include <sys/utsname.h>
42  #include <sys/socket.h>
43  #include <netinet/in.h>
44  #include <arpa/inet.h>
45  #include <netdb.h>

47  #include <string.h>
48  #include <stdlib.h>
49  #include <esl/inout.h>

51  #include <esl/c_portable.h>
52  #include <esl/ep_xopen.h>
53  #include <pthread.h>

55  // Rogue Wave includes
56  #include <rw/collect.h>
57  #include <rw/rwfile.h>

59  #include <rw/vstream.h>
    #include <rw/bintree.h>
```

```
61  #include <csc/cscomm.h>
62  #include <edmlink/edmlink_api.h>

64  #ifdef  __cplusplus
65  extern  "C" {
66  #endif

68  #include <restore/dispatch_daemon.h>
69  #include <restore/dispatch_protocol.h>
70  #include <restore/RestoreObjectID.h>
71  #include <restore/csc_Dispatch_Protocol_Service.h>
72  #include <restore/dispatch_protocol_service.h>
73  #include <restore/dispatch_protocol_client.h>
74  #include <dpService.h>

76  #ifdef  __cplusplus
77  }
78  #endif

80  #include <logging/logging.h>
81  #include <EDMDispatchLog.h>
82  #include <EDMDHandle.h>
83  #include <EDMDDHandleMgrApi.h>
84  #include <EDMSession.h>
85  #include <EDMccr.h>
86  #include <EDMutils.h>
87  #include <EDMDD_ddp.h>
88  #include <EDMDDcr_rstsvc.h>

90  pthread_cond_t cscPortRdy_cv = PTHREAD_COND_INITIALIZER;
91  pthread_mutex_t cscPortRdy_mutex = PTHREAD_MUTEX_INITIALIZER;
92  pthread_mutex_t G_serviceMtx;

94  static boolean32 print_error = TRUE;

96  /* Prototypes */
97  int ednrst_send_chndl_to_private_svc(int);
98  int ednrst_create_ddp_client_connection(
                     int,rpc_binding_handle_t **,EDMSession *);
99  int ednrst_send_uid_to_private_svc(int, EDMSession *);

101 /* Dispatch Protocol ifspec */
102 static rpc_if_handle_t DispatchDaemon_ifspec;
103 ElinkHandlePtr_ty  ElinkHandle;            /* Handle for svc object */

105 /***********************************************************************
106 **
107 ** Routine:     LockSvcMutex
108 **
109 ** Inputs:      None
110 **
111 ** Outputs:     None
112 **
113 ** Return Codes:
114 **              None
115 **
116 ** Purpose:     Lock the mutex for the service execution
117 **
118 **
119 ***********************************************************************/

121 static void
122 LockSvcMutex()
```

```
123  1  {
124  1      static boolean_ty first = TRUE;

126  1      if (first == TRUE)
127  2      {
128  2          first = FALSE;
129  2          pthread_mutex_init(&G_serviceMtx, NULL);
130  1      }

132  1      pthread_mutex_lock(&G_serviceMtx);
133     }
```

```
135     /****************************************************************************
136     **
137     ** Routine:   UnlockSvcMutex
138     **
139     ** Inputs:    None
140     **
141     ** Outputs:   None
142     **
143     ** Return Codes:
144     **            None
145     **
146     ** Purpose:   Unlock the mutex for service execution
147     **
148     ****************************************************************************
149     */
151     static void
152     UnlockSvcMutex()
153  1  {
154  1      pthread_mutex_unlock(&G_serviceMtx);
155     }
```

```
157    void *
158    DDRSTsvc_init(void *pSessObj)
159    {
160        int                     lrc;            /* Local Return Code            */
161        int                     fd1;
162        int                     fd2;
163        int                     status;
164        rpc_binding_handle_t *  bh=NULL;
165        EDMSession *            p_so;
166        // EDMDDHandle *        pEHandleObj;
167        DD_client_session_id    sID;
168        ElinkShellObjPtr_ty     *svc_rpc_h=NULL; /* X-Service RPC Handle */
169        unsigned char           ShellHandle;
170        ElinkTargetObjPtr_ty    TargetObjPtr;   /* Target object all functions  */
171        ElinkUserIdObjPtr_ty    UserIdObjPtr;   /* UserId object copy & shell   */
172        ElinkCmdObjPtr_ty       CmdObjPtr;      /* Shell command shell only     */
173        unsigned long           options = 0;    /* For ElinkNewServicelaunchObj */

176        // launch one service at a time.
177        //
178        //
179        LockSvcMutex();
180        pthread_mutex_lock( &cscPortRdy_mutex );

182        // Check to see that the EDMLINK handle didn't get trashed
183        //
184        //
185        if (ElinkHandle == NULL)
186        {
187            UnlockSvcMutex();
188            pthread_mutex_unlock( &cscPortRdy_mutex);
189            pthread_exit( NULL );
190        }

192        // Cast the input argument to its object type.
193        //
194        //
195        p_so = (EDMSession*)pSessObj;

197        // Construct EDM-Link target object so that EDM-Link will know what
198        // system we want to talk to.
199        //
200        //
202        TargetObjPtr = ElinkNewTargetObj( ElinkHandle,
203                                          "localhost" );

205        // EDM-Link should have called our callback DOMIELinkCallback which
206        // should have loaded DOMIHandle->ErrorBlock, so all we have to do
207        // now, is return.
208        if ( NULL == TargetObjPtr )
209        {
210            p_so -> setStatus(DD_SERVICE_FAILURE_NONEXEC);
211            UnlockSvcMutex();
212            pthread_mutex_unlock( &cscPortRdy_mutex);
213            pthread_exit( NULL );
214        }
```

```
218        //
219        // Construct EDM-Link user object.  We always want to run as root on the
220        //  target.     We are starting a private service that will run on an EDM.
221        // We know that we will be starting via the EDM-Link daemon and we
222        // always start using the root id.  Also, this will be a service, it needs
223        // to run as root and will have some intelligence in protecting itself in
224        // that there a limited things that it can do and the caller of the API
225        // will control what can be done.
226        //
227        // EDM-Link should have called our callback DOMIELinkCallback which
228        // should have loaded DOMIHandle->ErrorBlock, so all we have to do
229        // now, is return.
230        UserIdObjPtr = ElinkNewUserIdObj( ElinkHandle,
                                            TargetObjPtr,
                                            NULL,
                                            NULL );
232        if ( NULL == UserIdObjPtr )
233        {
234            (void) ElinkDestroyObj( ElinkHandle, TargetObjPtr );
235            p_so -> setStatus(DD_SERVICE_FAILURE_NONEXEC);
236            UnlockSvcMutex();
237            pthread_mutex_unlock( &cscPortRdy_mutex );
238            pthread_exit( NULL );
239        }

241        // Utilize the EDM-Link service launcher to physically startup the
242        // domain private service.  By convention, all private services can
243        // be found in /usr/epoch/service and have a suffix of pd.  The domain
244        // private service is: /usr/epoch/service/domainpd.
246        //
247        //
248        //
249        //
250        //
251        //
253        if (IsDebugOn())
254            options |= ELINK_SERVICE_DEBUG;      /* if we are debug, so will be R.Eng */

256        CmdObjPtr = ElinkNewServicelaunchObj( ElinkHandle,
257                                              TargetObjPtr,
258                                              "edmrestoreeng",
                                                UserIdObjPtr,
259                                              /* Domain private service */
260                                              options );

261        if ( NULL == CmdObjPtr )
262        {
263            (void) ElinkDestroyObj( ElinkHandle, TargetObjPtr );
264            (void) ElinkDestroyObj( ElinkHandle, UserIdObjPtr );
265            p_so -> setStatus(DD_SERVICE_FAILURE_NONEXEC);
266            UnlockSvcMutex();
267            pthread_mutex_unlock( &cscPortRdy_mutex);
268            pthread_exit( NULL );
269        }
270        // EDM-Link should have called our callback DOMIELinkCallback which
271        // should have loaded DOMIHandle->ErrorBlock, so all we have to do
272        // now, is return.
273    }
```

```
275  1
276  1    //
277  1    // Fire up Private Service via EDM-link API ElinkPrivateSvc. This
278  1    // physically starts the private service running.
279  1    //
280  1    lrc = ElinkPrivateSvc ( ElinkHandle,
281  1                            TargetObjPtr,
282  1                            UserIdObjPtr,
283  1                            CmdObjPtr,
284  1                            &fd1,
285  1                            &fd2,
286  1                            &ShellHandle ) ;
287  2
288  2    if ( -1 == lrc )
289  2    {
290  2        (void) ElinkDestroyObj( ElinkHandle, TargetObjPtr ) ;
291  2        (void) ElinkDestroyObj( ElinkHandle, UserIdObjPtr ) ;
292  2        (void) ElinkDestroyObj( ElinkHandle, CmdObjPtr ) ;
293  2        EDMDispatch_logent (
294  2            __FILE__, __LINE__, LOG_ERR, DDP_PRIVATE_SVC_FAILURE,
295  2            0, "ElinkPrivateSvc() failure") ;
296  2        pthread_mutex_unlock( &cscPortRdy_mutex) ;
297  1        pthread_exit( NULL ) ;
         }

299  1
300  1    //
301  1    // Extract the csc handle from the shell object. This handle
304  1    // is the restore service (restore API) rpc handle.
305  1    //
306  1    svc_rpc_h = (unsigned char*) calloc(1,CONNECT_HANDLE_SIZE) ;
307  1    if (svc_rpc_h == NULL )
308  1    {
309  1        EDMDispatch_logent(
310  2            __FILE__, __LINE__, LOG_ERR, DDP_NO_MEMORY,
311  1            0,"calloc() failure") ;
312  2        UnlockSvcMutex();
313  2        p_so -> setStatus(DD_SERVICE_FAILURE_NONEXEC);
314  2        pthread_mutex_unlock( &cscPortRdy_mutex) ;
315  2        pthread_exit( NULL ) ;
316  1    }
317  1    lrc = ElinkGetConnectHandle( ElinkHandle,
318  1                                 ShellHandle,
319  1                                 svc_rpc_h ) ;
320  1
321  2    if ( 0 != lrc )
322  2    {
323  2        EDMDispatch_logent(
324  2            __FILE__, __LINE__, LOG_ERR, DDP_GET_CONNECT_HANDLE_FAILURE,
325  2            0, "ElinkGetConnectHandle() failure") ;
326  2        UnlockSvcMutex();
327  2        p_so -> setStatus(DD_SERVICE_FAILURE_NONEXEC);
328  2        pthread_mutex_unlock( &cscPortRdy_mutex) ;
329  1        pthread_exit( NULL ) ;
         }

331  1    p_so -> setConnectionHandle((void *)svc_rpc_h);

333  1    p_so -> getSessionID(&sID);             // Get Unique Session id

335  1    //
336  1    // Issue message telling of Dispatch Daemon RDR port number.
337  1    //
```

```
339  1    if (IsDebugOn())
340  2    {
341  2        EDMDispatch_logent (
342  2            __FILE__, __LINE__, LOG_INFO, DDP_PORT_NUMBERS,
343  2            0, "PORT_INFO DispatchDaemon_ifspec
344  1                DDCCR) port#: %d",
                     DispatchDaemon_ifspec.portnum) ;
         }

346  1    //
347  1    // Unlock Port Rdy mutex so the Reader can listen.
348  1    //
349  1    pthread_mutex_unlock( &cscPortRdy_mutex) ;

351  1    //
352  1    // Tell the Dispatch Daemon Protocol Reader Thread to listen.
353  1    //
354  1    pthread_cond_signal(&cscPortRdy_cv);

356  1    //
357  1    // Inform the restore svc of dispatch protocol details ( port etc ...)
358  1    //
359  1    lrc = edmrst_send_chndl_to_private_svc(fd1);
360  1    if ( 0 != lrc )
361  1    {
362  2        EDMDispatch_logent(
363  2            __FILE__, __LINE__, LOG_ERR, DDP_CHANNEL_SEND_FAILURE,
364  2            0,"edmrst_send_chndl_to_private_svc(
                    ) failure") ;
365  2        UnlockSvcMutex();
366  2        p_so -> setStatus(DD_SERVICE_FAILURE_NONEXEC);
367  2        pthread_exit( NULL ) ;
368  1    }

370  1    //
371  1    // Send the Unique Session Id value.
372  1    //
373  1    lrc = edmrst_send_uid_to_private_svc(fd1,p_so);
374  1    if ( 0 != lrc )
375  2    {
376  2        EDMDispatch_logent(
377  2            __FILE__, __LINE__, LOG_ERR, DDP_SEND_UID_FAILURE,
378  2            0,"edmrst_send_uid_to_private_svc(
                    ) failure") ;
379  2        UnlockSvcMutex();
380  2        p_so -> setStatus(DD_SERVICE_FAILURE_NONEXEC);
381  2        pthread_exit( NULL ) ;
382  2    }

384  1    //
385  1    // Create the CCW service handle so we can respond to messages.
386  1    //
387  1    lrc = edmrst_create_ddp_client_connection(fd1, &bh, p_so);
388  1    if ( 0 != lrc )
389  2    {
390  2        EDMDispatch_logent(
391  2            __FILE__, __LINE__, LOG_ERR, DDP_CREATE_CLIENT_CONNECTION,
392  2            0,"edmrst_create_ddp_client_connection(
                    ) failure") ;
393  2        UnlockSvcMutex();
394  2        p_so -> setStatus(DD_SERVICE_FAILURE_NONEXEC);
395  2        pthread_exit( NULL ) ;
         }
```

```
396 1      }

398 1      //
399 1      // Insert handle object into Global list.
400 1      //
401 1      lrc = newHandleSet( &sID,
402 1                 fd1,
403 1                 fd2,
404 1                 bh,
405 1                 &ShellHandle,
406 1                 &status );
407 1
408 1      if ( 0 != lrc )
409 2      {
410 2         (void) free(svc_rpc_h);
411 2         EDMDispatch_logent(
412 2            __FILE__, __LINE__, LOG_ERR, DDP_HANDLE_INSERTION_ERROR,
413 2            status, "newHandleSet() failure");
414 2         UnlockSvcMutex();
415 1         pthread_exit( NULL );
417 1      }

418 1      //
419 1      // Let's clean up and set the status to RUNNING.
420 1      //
421 1      p_so -> setStatus(DD_SERVICE_RUNNING);
422 1      UnlockSvcMutex();
423 1      pthread_exit( NULL );
424 1      return( NULL );
           }
```

```
426   ** /*
427   ** ===============================================================================
428   ** Function:    edmrst_send_chndl_to_private_svc()
429   **
430   ** Description:
431   **
432   **
433   ** Returns:      0 Sucessful
434   **             -1 Read Failure
435   **             <0 Read less than exspected
436   **
437   ** ===============================================================================
438   */
439   int
440   edmrst_send_chndl_to_private_svc(int pipeToSvc)
441 1 {
442 1    auto int lrc=0;
         auto unsigned char *p_client_h=NULL;

444 1    //
445 1    // Isolate the connection handle from the server 'if_spec'.
446 1    // The IP/PORT are part of the created if_spec structure.
447 1    //
448 1    p_client_h = DispatchDaemon_ifspec:connect_handle_p;

450 1    //
451 1    // Write the handle to the service so it can contact me
452 1    //
453 1    lrc = edmrst_WrChannel(pipeToSvc,
454 1               p_client_h,
455 1               CONNECT_HANDLE_SIZE);
456 1    if ( CONNECT_HANDLE_SIZE != lrc )
457 2    {
458 2       (void) free(p_client_h);
459 2       EDMDispatch_logent( __FILE__, __LINE__, LOG_ERR, DDP_WRITE_CHANNEL,
460 2               0,"edmrst_WrChannel() Failure");
461 2       return(-1);
462 1    }

464 1    return(0);
465   }
```

```
467
468    /*
469    ** =========================================
470    ** Function:    edmrst_send_uid_to_private_svc()
471    **
472    ** Description:
473    **
474    **
475    ** Returns:
476    **      0  Sucessful
477    **     -1  Read Failure
478    **     <0  Read less than exepected
479    **
480    ** =========================================
481    */
482    int
483 1  edmrst_send_uid_to_private_svc(int pipeToSvc,
                                      EDMSession *pSessionObj)
484 1  {
486 1     auto int lrc=0;
487 1     auto DD_client_session_id uid;
488 1
489 1     // Write the handle to the service so it can contact me
490 1     //
491 1     pSessionObj -> getSessionID(&uid);
492 1     lrc = edmrst_WrChannel(pipeToSvc,
                                 (void*)&uid,
                                 sizeof(DD_client_session_id));
493 1     if ( (sizeof(DD_client_session_id)) != lrc )
494 2     {
495 2        EDMDispatch_logent( __FILE__, __LINE__, LOG_ERR,DDP_WRITE_CHANNEL,
                                 0,"edmrst_WrChannel() Failure");
496 2        return(-1);
497 2     }
498 1     return(0);
500 1  }
501
```

```
502
503    /*
504    ** =========================================
505    ** Function:    edmrst_create_ddp_client_connection()
506    **
507    ** Description:
508    **
509    **
510    ** Returns:
511    **      0  Sucessful
512    **     -1  Read Failure
513    **     <0  Read less than exepected
514    **
515    ** =========================================
516    */
517    int
518    edmrst_create_ddp_client_connection(int pipeToSvc,
                                           rpc_binding_handle_t **bh,
                                           EDMSession *p_so )
519 1  {
520 1     int lrc;
521 1     unsigned char *p_restore_service=NULL;
522 1     error_status_t status;
523 1     rpc_if_handle_t *p_psvc_ifspec=NULL;
          rpc_binding_handle_t *psvc_h=NULL;
525 1     //
526 1     // We now need to get the details from the restore service on
527 1     // how to connect from the dispatch daemon ccw to the restore
528 1     // service ccr. At this point, the restore service will be send-
529 1     // ing the restore service ccr handle information. The port / ip
530 1     // are the key information needed to create the ddp ccw handle.
531 1     //
532 1     lrc = edmrst_get_client_handle( pipeToSvc,&p_restore_service );
533 1     if ( 0 != lrc )
534 2     {
535 2        EDMDispatch_logent(
536 2           __FILE__, __LINE__, LOG_ERR,DDP_GET_CLIENT_HANDLE,
                0, "edmrst_get_client_handle() failure");
537 2        p_so -> setStatus(DD_SERVICE_FAILURE_NONEXEC);
538 2        return(-1);
539 1     }
541 1     //
542 1     // Create an ifspec from the handle
543 1     //
544 1     p_psvc_ifspec = (rpc_if_handle_t *)
545 1        calloc(1,sizeof(rpc_if_handle_t ));
546 1     if (p_psvc_ifspec == NULL)
547 2     {
548 2        EDMDispatch_logent(__FILE__, __LINE__, LOG_ERR,DDP_NO_MEMORY,
                                0,"ifspec calloc() failure");
549 2
550 2        return(-1);
551 1     }
553 1     lrc = csc_private_ifspec_init( p_restore_service,
                                        EDM_DISPATCH_PROTOCOL_CLIENT,
554 1                                   EDMPC_FUNCTIONS,
                                        p_psvc_ifspec,
                                        &status );
555 1
556 1     if ( 1 != lrc )
557 2     {
558 2        (void) free(p_psvc_ifspec);
559 2        EDMDispatch_logent(
560 2           __FILE__, __LINE__, LOG_ERR,DDP_IFSPEC_INIT_FAILURE,
                status, "csc_private_ifspec_init() failure");
561 2
562 2        return(-1);
563 2
```

```
564  1      }
566  1      if (IsDebugOn())
567  1      {
568  2          EDMDispatch_logent( __FILE__, __LINE__, LOG_INFO, DDP_PORT_NUMBERS,
569  2              0,"PORT_INFO p_psvc_ifspec(DDCCW) port#: %d",
570  2              p_psvc_ifspec->portnum);
571  1      }

573  1      psvc_h = (rpc_binding_handle_t *) calloc(1, sizeof(
574  1                       rpc_binding_handle_t));

575  1      //
576  1      // Using the connect handle (128 bytes) received from the restore
577  1      // service, connect to the restore service.
578  1      //
579  1      lrc = csc_connect_to_async_rpc_service( NULL,
580  1                       *p_psvc_ifspec,
581  1                       psvc_h,
582  1                       &status );

583  1      if ( 1 != lrc )
        {
584  2          EDMDispatch_logent(
585  2              __FILE__, __LINE__, LOG_ERR, DDP_PRIVATE_SVC_CONNECT_FAILURE,
586  2              status, "csc_connect_to_async_rpc_service(
587  2              ) Failure. Status is %d", status);
588  2          (void) free(p_psvc_ifspec);
589  1          (void) free(psvc_h);
                return(-1);
        }

591  1      *bh = psvc_h;
592  1      (void) free(p_psvc_ifspec);
594  1      return(0);
595    }
```

```
597  1  /*
598  1  ** =============================================================
599  1  **
600  1  ** Function:
601  1  ** Description:
602  1  **
603  1  **
604  1  ** Returns:     0   Sucessful
605  1  **             -1   Read Failure
606  1  **
607  1  ** =============================================================
608  1  */
609  1  int
610  1  EDMDDSvcInit()
611  1  {
612  1      struct hostent   *hp;
613  1      struct utsname   name;
614  1      error_status_t   csc_status;

616  1      int lrc = 0;

618  1      ElinkHandle = ElinkInitAPI(ELINK_SHELL_EDMLINK);
619  2      if (ElinkHandle == NULL)
620  2      {
621  1          return -1;
        }
623  1

624  1      //
625  1      // Initialize the ifspec specification from the private svc
626  1      // creation call. This call will output the DispatchDaemon_ifspec
627  1      //
628  1      lrc = csc_async_ifspec_init (&DispatchDaemon_ifspec,
                       CSC_IFSPEC_PRIVATE_TYPE,
629  2                       DP_PROGNUM,
630  2                       DP_VERSNUM,
631  1                       dispatch_func_p_t)&edm_dispatch_protocol_service_1_table,
632  1                       &csc_status);

635  1      if ( TRUE != lrc )
636  2      {
637  2          EDMDispatch_logent(
                   __FILE__, __LINE__, LOG_ERR, DDP_IFSPEC_INIT_FAILURE,
638  2              csc_status, "csc_async_ifspec_init() Failure" );
639  2          return(-1);
640  1      }

643  1      //
644  1      // We need the system name and ip for the if_spec.
645  1      //
646  1      uname( &name );
647  1      hp = gethostbyname( name.nodename );
648  1      if ( NULL == hp )
649  2      {
650  2          EDMDispatch_logent(
651  2              __FILE__, __LINE__, LOG_ERR, DDP_GETHOSTNAME_FAILURE,
652  2              0, "gethostbyname() failure" );
653  1          return -1;
        }
655  1      ( void ) memcpy( (char*) &DispatchDaemon_ifspec.ip_addr,
656  1                       hp->h_addr, hp->h_length );
```

```
658  1          //
659  1          // Register the callback functions.
660  1          //
661  1          lrc = csc_register_async_server_interface(
                         &DispatchDaemon_ifspec,
                         -1,
662  1                   edm_dispatch_protocol_service_1_table,
663  1                   edm_dispatch_protocol_service_1_nproc,
664  1                   &csc_status );
665  1
666  1
668  1          if ( TRUE != lrc )
                {
669  2              EDMDispatch_logent(
670  2                  __FILE__, __LINE__, LOG_ERR, DDP_REGISTER_SVC_FAILURE,
                         csc_status,
671  2                  "Failed to register asynchronous server interface." );
672  2
673  1              return -1;
                }
675  1          return 0;
676          }
```

```
DispDaemon_ccr                          14   (EDMDD_ccr.cc)
DispDaemon_ccw..............2                 (EDMDD_ccw.cc)
SendAbortRequestMessage     6                 (EDMDD_ccw.cc)
SendCloseRequestMessage.....7                 (EDMDD_ccw.cc)
SendConnectConfirmMessage   4                 (EDMDD_ccw.cc)
SendFinalStatsConfirmMessage....9            (EDMDD_ccw.cc)
SendPingRequestMessage      8                 (EDMDD_ccw.cc)
```

```
  1   /*
  2   ** ============================================================
  3   **
  4   **   Copyright 1996,1997 EMC Corporation
  5   **
  6   ** ============================================================
  7   */
  8   /*
  9   ** ============================================================
 10   **   EDMDD_ccw.c
 11   **
 12   ** Mission Statement: This is the entry point for the Control Channel
                            Writer thread.
 13   **
 14   **      Its main purpose is to write notifications or
                            progress to the
                            Dispatch Daemon.
 15   **
 16   ** Primary Data Acted On:
 17   **
 18   ** Compile-Time Options:
 19   **
 20   **      USE_SUNRPC - Compile source with sunrpc support.  If
                           not set, assume DCE support.
 21   **
 22   ** Basic idea here: Module for Control Channel Writer thread.
 23   ** ============================================================
 24   */
 25   /*
 26   ** ============================================================
 27   ** The following provides an RCS id in the binary that can be located
         with the what(1) utility.  The intent is to keep this short.
 28   ** ============================================================
 29   */
 30   #if !defined(lint)
 31   static char
 32       RCS_id [] = "@(#)$RCSfile: EDMccw.c,v $ "
 33                   "$Revision: 1.23 $"
                      "$Date: 1997/02/06 20:49:15 $" ;
 34   #endif

 36   /* #define _POSIX_SOURCE    unable to compile with this define set */
 37   /* #define _XOPEN_SOURCE    unable to compile with this define set */

 39   #include <c_portable.h>
 40   #include <esl/ep_xopen.h>
 41   #include <esl/inout.h>

 43   #include <pthread.h>
 44   #include <csc/cscomm.h>

 46   #include <rw/collect.h>

 48   #ifdef __cplusplus
 49   extern "C" {
 50   #endif

 52   #include <restore/dispatch_protocol.h>
 53   #include <restore/dispatch_protocol_client.h>
 54   #include <restore/csc_Dispatch_Protocol_Client.h>

 56   #ifdef __cplusplus
 57   }
```

```
 58   #endif

 60   #include <EDMDD_ccw.h>
 61   #include <EDMDD_ddp.h>
 62   #include <EDMReturnMessageApi.h>
 63   #include <EDMccw.h>
 64   #include <EDMutils.h>
 65   #include <EDMDDHandleMgrApi.h>
 66   #include <EDMDispatchSession.h>
 67   #include <logging/logging.h>
 68   #include <EDMDispatchlog.h>

 70   extern ElinkHandlePtr_ty ElinkHandle;

 72   // Internal Routines
 73   static int SendConnectConfirmMessage(
 74       DD_client_session_id*,rpc_binding_handle_t*);
 75   static int SendAbortRequestMessage(
          DD_client_session_id*,rpc_binding_handle_t*);
 76   static int SendCloseRequestMessage(
          DD_client_session_id*,rpc_binding_handle_t*);
 77   static int SendPingRequestMessage(
          DD_client_session_id*,rpc_binding_handle_t*);
 79   static int SendFinalStatsConfirmMessage(
          DD_client_session_id*,rpc_binding_handle_t*);

 80   void *
 81   DispDaemon_ccw(void *buf)
 82   {
 83       DD_client_session_id sid;
 84       rpc_binding_handle_t *client_h_p=NULL;
 85       int sstatus=0;
 86       int status=0;
 87       int ResponseMessage=0;
 89       int rc=0;

 90       for( ; ; )
 91       {
 92           /* Monitor the event queue for messages to send. */
 93           rc = PopResponseMessage(&ResponseMessage,
 94                                   &sid,
 95                                   &client_h_p,
 96                                   &status);

 97           if ( -1 == rc )
 98           {
 99               sleep( DISPATCH_CCW_SLEEP );
100               continue;
102           }

104           rc = GetSessionStatus(&sid, &sstatus, &status);

105           if ((rc != 0) && (ResponseMessage != dp_ping_request))
106           {
107               (void) EDMDispatch_logent( __FILE__, __LINE__, LOG_ERR,
108                      DDP_GET_SESSION_STATUS_FAILURE, status,
109                      "Can't retrieve status for session "
110                      "<%ld:%ld> -- drop message.",
111                      sid.high, sid.low);
                    continue;
113           }

114           if (sstatus == DD_SERVICE_FAILURE_NONEXEC || sstatus ==
115               DD_SERVICE_FAILURE_EXEC ||
116               sstatus == DD_SERVICE_FAILURE_PERMS)
                {
                    (void) EDMDispatch_logent( __FILE__, __LINE__, LOG_INFO,
```

```
117  3            "DDP_DROP_MESSAGE, 0,
118  3            "Session <%ld:%ld> failed to start - drop message. ",
119  3            sid.high, sid.low);
120  2
121  2            continue;
         }

     switch( ResponseMessage )
     {
         /* execute the callback that will process this message */

127  3        case dp_connect_confirm:
128  3            rc = SendConnectConfirmMessage(
129  3                 &sid, client_h_p);
130  3            break;

132  3        case dp_abort_request:
133  3            rc = SendAbortRequestMessage(
134  3                 &sid, client_h_p);
135  3            break;

136  3        case dp_close_request:
137  3            rc = SendCloseRequestMessage(
138  3                 &sid, client_h_p);
139  3            break;

140  3        case dp_event_confirm:
141  3            // No confirm needed for this message
142  3            break;

143  3        case dp_progress_confirm:
144  3            // No confirm needed for this message
145  3            break;

146  3        case dp_final_stats_confirm:
147  3            rc = SendFinalStatsConfirmMessage(
148  3                 &sid, client_h_p);
149  3            break;

     case dp_ping_request:
         rc = SendPingRequestMessage(&sid, client_h_p);
         break;

     default:
         EDMDispatch_logent(
             __FILE__, __LINE__, LOG_ERR, DDP_INVALID_MESSAGE,
             0,"Invalid message type received.");

         break;
     }

     /* Check for a shutdown setting */
150  2    if (rc != 0)
152  2        sleep(1);
153  2    }
155  2    }  /* End of forever loop */
156  1    return((void*)0);
157  1 }  /* End of DispDaemon_ccw() */
158
```

```
160  2  //
161  2  // Function: SendConnectConfirmMessage()
162  2  // Description:
163  2  //     Send the confirm connect message to the
164  2  //     dispatch daemon.
165  2  //
166  1  static int
167     SendConnectConfirmMessage(
168  1      DD_client_session_id *ssid, rpc_binding_handle_t *clnt_p )
169  1  {
170  1      int *rc = NULL;
171  1      int lrc = 0;
172  1      int savedrc = 0;
173  1      int status = 0;
174  1      int sstatus = 0;
175  1      DP_connect_confirm_msg *msg_p=NULL;
176  1
177  2      if (clnt_p != NULL)
178  2      {
179  2          msg_p = (DP_connect_confirm_msg*)
180  2              calloc(1, sizeof(DP_connect_confirm_msg));
181  2          msg_p->sid.high = ssid->high;
                msg_p->sid.low  = ssid->low;

183  2          rc = dp_connect_confirm_1(msg_p, *clnt_p);

185  2          if (IsDebugOn())
186  3          {
187  2              (void) EDMDispatch_logent( __FILE__, __LINE__, LOG_INFO,
188  3                  DDP_SENDING_MESSAGE, 0,
189  3                  "Sending dp_connect_confirm_1 message.");
190  2          }

191  1          free( msg_p );
192  2      }
193  2      else
194  2      {
195  2          /* Get the csc_binding_handle associated with this sid */
197  2          rpc_binding_handle_t *client_handle_p = NULL;

198  2          lrc = GetCSCHandle(ssid,
199  2                   &client_handle_p,
200  2                   &status );

202  2          if (0 != lrc)
203  2          {
204  2              EDMDispatch_logent(
                        __FILE__, __LINE__, LOG_ERR, DDP_GET_CSC_HANDLE_FAILURE, status,
                        "GetCSCHandle failed.");

205  3              savedrc = lrc;
206  2          }

207  2          /* Push message to send onto the queue */
209  2          lrc = PushResponseMessage((int) dp_connect_confirm,
210  2                   *ssid,
211  2                   client_handle_p,
212  2                   &status);

213  2          if (0 != lrc)
214  2          {
215  2              EDMDispatch_logent(
216  3                  __FILE__, __LINE__, LOG_ERR, DDP_PUT_RESPONSE_FAILURE, status,
217  3                  "PushResponseMessage failed.");

218  3              savedrc = lrc;
219  2          }
         }
```

```
221  2        lrc = savedlrc;

223  2        return lrc;
224  1    }

226  1    return(0);
227  }
```

```
229      //
230      // Function: SendAbortRequestMessage()
231      // Description:
232      //    Send a abort request to a restore service.
233      //
234      static int
235      SendAbortRequestMessage(
236  1        DD_client_session_id *ssid, rpc_binding_handle_t *clnt_p )
237  1    {
238  1        int *rc;
             DP_abort_request_msg *msg_p=NULL;

240  1        msg_p = (DP_abort_request_msg*)
241  1            calloc(1, sizeof(DP_abort_request_msg));
242  1        msg_p->ssid.high = ssid->high;
243  1        msg_p->ssid.low = ssid->low;
244  1        rc = dp_abort_request_1(msg_p, *clnt_p);

246  1        if (IsDebugOn())
247  2        {
248  2            (void) EDMDispatch_logent( __FILE__, __LINE__, LOG_INFO,
                     DDP_SENDING_MESSAGE, 0,
                     "Sending dp_abort_request_1 message.");
249  2
250  2        }
251  1    }

253  1        free( msg_p );
254  1        return(0);
255      }
```

```
257     //
258     // Function: SendCloseRequestMessage()
259     // Description:
260     //     Send a close request to a restore service.
261     //
262     static int
263     SendCloseRequestMessage(
            DD_client_session_id *ssid, rpc_binding_handle_t *clnt_p )
264     {
265  1     int *rc;
266  1     DP_close_request_msg *msg_p=NULL;

268  1     msg_p = (DP_close_request_msg*)
269  1          calloc(1, sizeof(DP_close_request_msg));
270  1     msg_p->ssid.high = ssid->high;
271  1     msg_p->ssid.low  = ssid->low;
272  1     rc = dp_close_request_1(msg_p, *clnt_p);

274  1     if (IsDebugOn())
275  2     {
276  2        (void) EDMDispatch_logent( __FILE__, __LINE__, LOG_INFO,
                 DDP_SENDING_MESSAGE, 0,
                 "Sending dp_close_request_1 message.");
279  1     }

281  1     free( msg_p );
282  1     return(0);
283     }
```

```
285     //
286     // Function: SendPingRequestMessage()
287     // Description:
288     //     Send a ping request to a restore service.
289     //
290     static int
291     SendPingRequestMessage(
            DD_client_session_id *ssid, rpc_binding_handle_t *clnt_p )
292     {
293  1     int *rc;
294  1     DP_ping_request_msg *msg_p=NULL;

296  1     msg_p = (DP_ping_request_msg*)
297  1          calloc(1, sizeof(DP_ping_request_msg));
298  1     msg_p->ssid.high = ssid->high;
299  1     msg_p->ssid.low  = ssid->low;
300  1     rc = dp_ping_request_1(msg_p, *clnt_p);

302  1     if (IsDebugOn())
303  2     {
304  2        (void) EDMDispatch_logent( __FILE__, __LINE__, LOG_INFO,
                 DDP_SENDING_MESSAGE, 0,
                 "Sending dp_ping_request_1 message.");
307  1     }

309  1     free( msg_p );
310  1     return(0);
311     }
```

```
313
314    //
315    // Function: SendFinalStatsConfirmMessage()
316    // Description:
317    //     Send a ping request to a restore service.
318    //
319    static int
320    SendFinalStatsConfirmMessage(
321        DD_client_session_id *ssid, rpc_binding_handle_t *clnt_p )
322    {
323        int status, *rc;
324        int out = -1, err = -1;
325        int ret = 0;
327        rpc_binding_handle_t *ptr;
328        DP_final_stats_confirm_msg *msg_p=NULL;

329        if (clnt_p != NULL)
330        {
331            msg_p = (DP_final_stats_confirm_msg*)
332                calloc(1, sizeof(DP_final_stats_confirm_msg));
333            msg_p->ssid.high = ssid->high;
335            msg_p->ssid.low  = ssid->low;
336            rc = dp_final_stats_confirm_1(msg_p, *clnt_p);

337            if (IsDebugOn())
338            {
339                (void) EDMDispatch_logent( __FILE__, __LINE__, LOG_INFO,
340                    DDP_SENDING_MESSAGE, 0,
                       "Sending dp_final_stats_confirm_1
                       message.") ;
342            }

343            free( msg_p ) ;
345        }

346        ret = removeSession(ssid, &status) ;
347        if (ret == -1)
348        {
349            (void) EDMDispatch_logent( __FILE__, __LINE__, LOG_ERR,
350                DDP_REMOVE_SESSION_FAILURE, status,
                   "Failure removing session instance from list.
                   ") ;
351        }

354        ret = getHandleSet(ssid, &out, &err, &status) ;
355        if (ret == -1)
356        {
357            (void) EDMDispatch_logent( __FILE__, __LINE__, LOG_ERR,
358                DDP_GET_HANDLE_SET_FAILURE, status,
                   "Failure getting session handles from list.
                   ") ;
359        }

361        if (out != -1 && err != -1)
362        {
363            close(out);
364            close(err);
365        }

367        ret = deleteHandleSet(ssid, &ELinkHandle, &status);
368        if (ret == -1)
369        {
370            (void) EDMDispatch_logent( __FILE__, __LINE__, LOG_ERR,
371                DDP_DELETE_HANDLE_SET_FAILURE, status,
                   "Failure removing session handles from list.
                   ") ;
```

```
373    }

375    return(0);
376    }
```

```
 1   /*
 2   **
 3   ** ================================================================
 4   ** ================================================================
 5   */
 6   /*
 7   **
 8   ** EDMDD_ccr.cc
 9   **
10   ** Mission Statement: This is the entry point for the Control Channel
11                         Reader thread. Its main purpose is to read asynchronous
                          messages from the Dispatch Daemon.
12   ** ================
13   ** ============
14   ** ==========
15   ** Primary Data Acted On:
16   **
17   ** Compile-Time Options:
18   **
19   **    USE_SUNRPC - Compile source with sunrpc support.  If
                        not set, assume DCE support.
20   **
21   **
22   ** Basic idea here: Module for Control Channel Reader thread.
23   ** =========================
24   */
25   /*
26   **
27   ** The following provides an RCS id in the binary that can be located
28   ** with the what(1) utility. The intent is to keep this short.
29   ** =========================
30   */
31   #if !defined(lint)
32   static char   RCS_id [] = "@(#)$RCSfile: EDMccr.c,v $ "
33                             "$Revision: 1.23 $ "
34                             "$Date: 1997/02/06 20:49:15 $ ";
35   #endif

37   /* #define _POSIX_SOURCE        unable to compile with this define set */
38   /* #define _XOPEN_SOURCE        unable to compile with this define set */

40   #include <signal.h>
41   #include <esl/c_portable.h>
42   #include <esl/ep_xopen.h>
43   #include <esl/inout.h>

45   #include <pthread.h>
46   #include <logging/logging.h>
47   #include <EDMDD_ddp.h>
48   #include <EDMDD_ccr.h>
49   #include <csc/cscomm.h>
50   #include <EDMDispatchLog.h>

52   static void halt_service(int);
53   static boolean32 print_error = TRUE;

55   extern pthread_cond_t cscPortRdy_cv;
56   extern pthread_mutex_t cscPortRdy_mutex;

58   void *
```

```
 59    DispDaemon_ccr(void *buff)
 60    {
 61  1     int lrc=0;
 62  1     error_status_t   status;
 63  1     rpc_if_handle_t  if_spec;
 64  1     struct esl_timeval timeout = {5,0};

 66  1    //
 67  1    // Wait for transient thread to tell me there is something to listen on.
 68  1    //
 69  1     pthread_mutex_lock( &cscPortRdy_mutex );
 70  1     pthread_cond_wait( &cscPortRdy_cv, &cscPortRdy_mutex );
 71  1     pthread_mutex_unlock( &cscPortRdy_mutex );

 73  1    /*
 74  1    ** =========================
 75  1    ** Let begin to listen for requests.
 76  1    ** =========================
 77  1    */
 78  1     for(;;)
 79  2     {
 80  2        lrc = csc_async_server_listen( (esl_timeval*)&timeout, &status);
 81  2        if ( 1 != lrc )
 82  3        {
 83  3           EDMDispatch_logent(
 84  3               __FILE__, __LINE__, LOG_ERR, DDP_FAILED_LISTEN,
 85  2               0, "Bad returned from listen." );
 86  2        }
 87  3        if ( 1 == lrc )
 88  3        {
 89  3           EDMDispatch_logent(
 90  2               __FILE__, __LINE__, LOG_INFO, DDP_LISTEN_TIMEOUT,
 91  1               0,"listen() timedout." );
 92          }
 93  1     } /* End of while loop */
 94  1    /*
 95  1    ** =========================
 96  1    ** Unregister our service upon exit request.
 97  1    ** =========================
 98  1    */
 99  1     lrc = csc_unregister_async_server_interface(&if_spec, &status);
100  1     EDMDispatch_logent( __FILE__, __LINE__, LOG_INFO, DDP_UNREGISTER_SVC,
101  1                0, "Returned from unregister service." );
102  1     return((void*)0);
        }
```

EDMDD_ccr.cc 3

EDMDD_ccr.cc 4

```
 1   /*
 2   ** Copyright 1996,1997 EMC Corporation
 3   */

 6   /*
 7   ** EDMDDHandle.cc
 8   **
 9   ** Mission Statement:  file that contains the Handle class methods
10   **
11   ** Primary Data Acted On:
12   **
13   ** Compile-Time Options:
14   **
15   ** Basic idea here:
16   **
17   **     The Handle object is a container which holds a
18   **     set of handles for each running service.
     */

20   #if !defined(lint)
21   static char     RCS_id [] = "@(#)$RCSfile: EDMDDHandle.cc,v $ "
22                   "$Revision: 1.0 $ "
                     "$Date: 1997/02/06 20:49:15 $" ;
24   #endif

26   #include <esl/c_portable.h>
27   #include <esl/ep_xopen.h>
28   #include <esl/inout.h>

30   #include <string.h>
31   #include <stdlib.h>

33   // Rogue Wave includes
34   #include <rw/collect.h>
35   #include <rw/rwfile.h>
36   #include <rw/vstream.h>

38   #include <csc/cscomm.h>

40   #include <edmlink/edmlink_api.h>
41   #include <restore/RestoreObjectID.h>
42   #include <restore/dispatch_daemon.h>
43   #include <EDMDDHandle.h>

45   // Needed for rogue wave linked list manager.
46   // 413 is the object ID.
47   RWDEFINE_COLLECTABLE(EDMDDHandle, EDMDHANDLE)

49   /****************************************************************
50   **
51   ** Routine:    EDMDDHandle constructor
52   **
53   ** Inputs:     None
54   **
55   ** Outputs:    None
56   **
57   ** Return Codes:
58   **             None
59   **
60   ** Purpose:    Initializes the Handle class by resetting the internal
61   **             data.
62   **
63   ****************************************************************
```

```
65   EDMDDHandle::EDMDDHandle()
66   {
67       rpcBD = NULL;
68       shellHd = NULL;
69       stdoutPipe = 0;
70       stderrPipe = 0;

72       memset(&sessionID, 0, sizeof(sessionID));
73   }

75   /****************************************************************
76   **
77   ** Routine:    EDMDDHandle constructor
78   **
79   ** Inputs:     rpc_binding_handle_t bh - the client handle to use for the
                                                                 dispatch protocol
80   **             int stdoutpipe - the stdout descriptor of the service
81   **             int stderrpipe - the stderr descriptor of the service
82   **             DD_client_session_id *sess - the session ID of the
                                                                 service
83   **
84   ** Outputs:    None
85   **
86   ** Return Codes:
87   **             None
88   **
89   ** Purpose:    Initializes the internals of the Handle class.
90   **
91   ****************************************************************
92   */

95   EDMDDHandle::EDMDDHandle(
             IN rpc_binding_handle_t *bh, IN DD_client_session_id *sess,
             IN int stdoutpipe, IN int stderrpipe)
97   {
98       rpcBD = bh;
99       stdoutPipe = stdoutpipe;
100      stderrPipe = stderrpipe;

102      if (sess != NULL)
103          memcpy(&sessionID, sess, sizeof(DD_client_session_id));
104  }

106  /****************************************************************
107  **
108  ** Routine:    EDMDDHandle destructor
109  **
110  ** Inputs:     None
111  **
112  ** Outputs:    None
113  **
114  ** Return Codes:
115  **             None
116  **
117  ** Purpose:    Doesn't really do anything but seems to be a requirement
118  **             for the linked list manager.
119  **
120  ****************************************************************
121  */
122
```

```
123 1      }
124 1    }
126      /*******************************************************
127      **
128      ** Routine:    compareTo
129      **
130      ** Inputs:     RWCollectable *c - a pointer to the base class type which
131      **                            you can then cast and compare.
132      **
133      ** Outputs:    None
134      **
135      ** Return Codes:
136      **      int    -   returns numbers like qsort compare (-1, 0, 1)
137      **
138      ** Purpose:  Compare using the auxproc PID.
139      **
140      *******************************************************
141     */
143 int
144 EDMDDHandle::compareTo(IN const RWCollectable *c) const
145 1 {
146 1     EDMDDHandle *localhandle = (EDMDDHandle *) c;
148 1     if (localhandle == NULL)
149 1         return -1;
151 1     if (localhandle -> sessionID.high == sessionID.high &&
152 1         localhandle -> sessionID.low == sessionID.low)
153 1         return 0;
154 1     return (localhandle -> sessionID.high > sessionID.high ||
155 1         (localhandle -> sessionID.high == sessionID.high &&
156 1          localhandle -> sessionID.low > sessionID.low) ? 1 : -1;
157 }

159 /*******************************************************
160 **
161 ** Routine:    isEqual
162 **
163 ** Inputs:     RWCollectable *c - a pointer to the base class type which
164 **                            you can then cast and compare.
164 **
165 ** Outputs:    None
166 **
167 ** Return Codes:
168 **      RWBoolean  -   TRUE or FALSE
169 **
170 ** Purpose:  Compare session IDs to find which session needs service.
171 **
172 **
173
174 */
176 RWBoolean
177 EDMDDHandle::isEqual(IN const RWCollectable *c) const
178 1 {
179 1     EDMDDHandle *localhandle = (EDMDDHandle *) c;
181 1     if (localhandle == NULL)
182 1         return FALSE;
184 1     if (localhandle -> sessionID.high == sessionID.high &&
```

```
185 1         localhandle -> sessionID.low == sessionID.low)
186 1         return TRUE;
187 1     else
188 1         return FALSE;
189 1 }
191 /*******************************************************
192 ** Routine:    hash
193 **
194 ** Inputs:     None
195 **
196 ** Outputs:    None
197 **
198 ** Return Codes:
199 **      unsigned  -   returns time started
200 **
201 ** Purpose:  Returns unique value, in this case auxproc pid.
202 **
203 **
204 *******************************************************
205 */
207 unsigned
208 EDMDDHandle::hash() const
209 1 {
210 1     return (unsigned) sessionID.low;
211 1 }
213 /*******************************************************
214 ** Routine:    saveGuts
215 **
216 ** Inputs:     RWFile    f - File pointer where data will be saved.
217 **
218 ** Outputs:    None
219 **
220 ** Return Codes:
221 **      None
222 **
223 ** Purpose:  Save class internal data to a file.
224 **
225 **
226 *******************************************************
227 */
229 void
230 EDMDDHandle::saveGuts(IN RWFile &f)
231 1 {
232 1     // Save parent class data too
233 1     RWCollectable::saveGuts(f);
235 1     // Left as an example
236 1 }
238 /*******************************************************
239 **
240 ** Routine:    saveGuts
241 **
242 ** Inputs:     RWvostream strm - stream to write internal data to.
243 **
244 ** Outputs:    None
245 **
```

```
246   **  Return Codes:
247   **      None
248   **
249   **  Purpose:  Save class data to a stream.
250   **
251       ****************************************************/
252   */
254   void
255   EDMDDHandle::saveGuts(IN RWvostream &strm)
256 1 {
257 1     // Save parent class data too
258 1     RWCollectable::saveGuts(strm);
260 1     // Left as an example
261 1 }
263   /***************************************************
264   **
265   **  Routine:  restoreGuts
266   **
267   **  Inputs:   RWFile f - file to read internal data from.
268   **
269   **  Outputs:  None
270   **
271   **  Return Codes:
272   **      None
273   **
274   **  Purpose:  Restores an instance of the Handle class by reading the
275   **            data from the passed in file.
276   **
277       ***************************************************/
278   */
280   void
281   EDMDDHandle::restoreGuts(IN RWFile &f)
282 1 {
283 1     // Restore parent data too
284 1     RWCollectable::restoreGuts(f);
286 1     // Left as an example
287 1 }
289   /***************************************************
290   **
291   **  Routine:  restoreGuts
292   **
293   **  Inputs:   RWvistream strm - stream to read internal data from.
294   **
295   **  Outputs:  None
296   **
297   **  Return Codes:
298   **      None
299   **
300   **  Purpose:  Restores an instance of the Handle class by reading the
301   **            data from the passed in stream.
302   **
303       ***************************************************/
304   */
```

```
306   void
307   EDMDDHandle::restoreGuts(IN RWvistream &strm)
308 1 {
309 1     // Restore parent data too
310 1     RWCollectable::restoreGuts(strm);
312 1     // Left as an example
313 1 }
315   /***************************************************
316   **
317   **  Routine:  binaryStoreSize
318   **
319   **  Inputs:   None
320   **
321   **  Outputs:  None
322   **
323   **  Return Codes:
324   **      RWspace count - file size of class written to disk in
325   **                      bytes
326   **  Purpose:  Returns the size of class if it were stored on disk.
327   **
328       ***************************************************/
329   */
331   RWspace
332   EDMDDHandle::binaryStoreSize() const
333 1 {
334 1     RWspace count = RWCollectable::binaryStoreSize();
335 1     return count;
336 1 }
338   /***************************************************
339   **
340   **  Routine:  getSessionID
341   **
342   **  Inputs:   None
343   **
344   **  Outputs:  None
345   **
346   **  Return Codes:
347   **      DD_client_session_id sessionID - the session ID
348   **
349   **  Purpose:  Returns the ID of the session the object belongs to.
350   **
351       ***************************************************/
352   */
354   DD_client_session_id
355   EDMDDHandle::getSessionID()
356 1 {
357 1     return sessionID;
358 1 }
360   /***************************************************
361   **
362   **  Routine:  setSessionID
363   **
364   **  Inputs:   DD_client_session_id ID - the session ID associated with
```

```
                                this object
365  **
366  **
367  **  Outputs:  None
368  **
369  **
370  **  Return Codes:
371  **      None
372  **
373  **  Purpose:  Sets the ID of the session the object belongs to.
374  **
375  *****************************************************
     */
377  void
378  EDMDDHandle::setSessionID(DD_client_session_id ID)
379  {
380      sessionID = ID;
381  }

383  /*****************************************************
384  **
385  **  Routine:  getBindingHandle
386  **
387  **  Inputs:  None
388  **
389  **
390  **  Outputs:  None
391  **
392  **  Return Codes:
393  **      rpc_binding_handle_t rpcBD - the binding handle for the client
                side of the dispatch protocol
394  **
395  **  Purpose:  Returns the binding handle for the client side of the
                dispatch
396  **            protocol.
397  **
398  *****************************************************
399  */
401  rpc_binding_handle_t *
402  EDMDDHandle::getBindingHandle()
403  {
404      return rpcBD;
405  }

407  /*****************************************************
408  **
409  **  Routine:  setBindingHandle
410  **
411  **  Inputs:  rpc_binding_handle_t *bh - the binding handle to use for
                this
412  **              service
413  **
414  **  Outputs:  None
415  **
416  **  Return Codes:
417  **      None
418  **
419  **  Purpose:  Sets the binding handle of the session the object belongs
                to.
420  **
421  *****************************************************
```

```
422  */
424  void
425  EDMDDHandle::setBindingHandle(rpc_binding_handle_t *bh)
426  {
427      rpcBD = bh;
428  }

430  /*****************************************************
431  **
432  **  Routine:  getShellHandle
433  **
434  **  Inputs:  None
435  **
436  **
437  **  Outputs:  None
438  **
439  **  Return Codes:
440  **      ElinkShellObjPtr_ty shellHd - the shell handle for the client
                side of the dispatch protocol
441  **
442  **  Purpose:  Returns the shell handle for the client side of the
                dispatch
443  **            protocol.
444  **
445  **
446  *****************************************************
     */
449  ElinkShellObjPtr_ty *
450  EDMDDHandle::getShellHandle()
451  {
452      return &shellHd;
     }

454  /*****************************************************
455  **
456  **  Routine:  setShellHandle
457  **
458  **  Inputs:  ElinkShellObjPtr_ty *bh - the shell handle to use for
                this
459  **              service
460  **
461  **  Outputs:  None
462  **
463  **  Return Codes:
464  **      None
465  **
466  **  Purpose:  Sets the shell handle of the session the object belongs
                to.
467  **
468  *****************************************************
469  */
471  void
472  EDMDDHandle::setShellHandle(ElinkShellObjPtr_ty *bh)
473  {
474      shellHd = *bh;
475  }

477  /*****************************************************
478  **
```

```
479  **  Routine:  getStdoutPipe
480  **
481  **  Inputs:   None
482  **
483  **  Outputs:  None
484  **
485  **  Return Codes:
486  **      int stdoutFD - the stdout descriptor of the service
487  **
488  **  Purpose:  Returns the stdout handle of the service.
489  **
490  ****************************************************************
491  */
492  int
493  EDMDDHandle::getStdoutPipe()
494  1 {
495  1     return stdoutPipe;
496  1 }
497
498  /*****************************************************************
499  ******************************************************************
500  **
501  **  Routine:  setStdoutPipe
502  **
503  **  Inputs:   int handle - the stdout handle of the private service
504  **
505  **  Outputs:  None
506  **
507  **  Return Codes:
508  **      None
509  **
510  **  Purpose:  Sets the stdout handle of the private service.
511  **
512  **
513  ******************************************************************
514  void
515  EDMDDHandle::setStdoutPipe(int handle)
516  1 {
517  1     stdoutPipe = handle;
518  1 }
519
520  /*****************************************************************
521  ******************************************************************
522  **
523  **  Routine:  getStderrPipe
524  **
525  **  Inputs:   None
526  **
527  **  Outputs:  None
528  **
529  **  Return Codes:
530  **      int stderrPipe - the stderr descriptor of the service
531  **
532  **  Purpose:  Returns the stderr descriptor of the service.
533  **
534  ******************************************************************
535  */
536  int
537  EDMDDHandle::getStderrPipe()
538  {
539  1 {
```

```
540  1     return stderrPipe;
541  }
542
543  /*****************************************************************
544  **
545  **  Routine:  setStderrPipe
546  **
547  **  Inputs:   int handle - the stderr handle of the service
548  **
549  **  Outputs:  None
550  **
551  **  Return Codes:
552  **      None
553  **
554  **  Purpose:  Sets the stderr handle of the service.
555  **
556  ******************************************************************
557  */
558  void
559  EDMDDHandle::setStderrPipe(int handle)
560  {
561  1     stderrPipe = handle;
562  1 }
563
```

EDMDDHandle.cc 11

EDMDDHandle.cc 12

```
1    /*
2    ** Copyright 1996,1997 EMC Corporation
3    *
     *
6    ** EDMDDHandleMgrApi.cc
7    *
8    *
9    * Mission Statement:  An API to manage the handle sets/objects
10   *
11   * Primary Data Acted On:
12   *
13   * Compile-Time Options:
14   *
15   * Basic idea here:
16   *
17   *          This API manages the handle sets.  Multiple threads
18   *          need access to the handles to do IO.
19   *
20   *          Each time an fd_set is modified or used we lock
21   *          a mutex to make sure access is serialized.
     */
23   #if !defined(lint)
24   static char   RCS_id [] = "@(#)$RCSfile: EDMDDHandleMgrApi.cc,v $ "
25                             "$Revision: 1.0 $ "
26                             "$Date: 1997/02/06 20:49:15 $ " ;
27   #endif

27   #include <esl/c_portable.h>
29   #include <esl/ep_xopen.h>
30   #include <esl/inout.h>
31
33   #include <stdlib.h>
34   #include <sys/types.h>
35   #include <sys/time.h>
36   #include <pthread.h>
36
38   #include <EDMDDHandleMgrApi.h>

40   // The tree that we keep handles in.  We could have used any
41   // Rogue Wave object but I decided to use the tree.

43   static RWBinaryTree G_handleTree;

45   // These are the fd_sets managed on behalf of the user.  The
46   // Modified sets are changed any time new handles are added
47   // and the others are the actual copies given to the user.
48   // This allows the manager to add handles from 1 thread without
49   // directly effecting a select call made in another thread.

51   fd_set    stdoutSetModified,
52             stdoutSet,
53             stderrSetModified,
54             stderrSet;

56   // These are values for the highest number handle that is part of a
57   // given set.  Keep in mind that 1 has to be added to this number to
58   // select on the highest handle used.

60   int highStdout = 0,
61       highStderr = 0;

63   static pthread_mutex_t   G_fdSetMtx = PTHREAD_MUTEX_INITIALIZER;
64   static pthread_mutex_t   G_handleTreeMtx = PTHREAD_MUTEX_INITIALIZER;
```

```
66   /*********************************************************************
67   **
68   ** Routine:  initFdSets
69   **
70   ** Inputs:   None
71   **
72   ** Outputs:  None
73   **
74   ** Return Codes:
75   **           None
76   **
77   ** Purpose:  Initialize fd sets and mutex.
78   **
79   **********************************************************************
80   */

82   static void
83   initFdSets()
84   {
85       FD_ZERO(&stdoutSetModified);
86       FD_ZERO(&stdoutSet);
87       FD_ZERO(&stderrSetModified);
88       FD_ZERO(&stderrSet);

90       pthread_mutex_init(&G_fdSetMtx, NULL);
91   }
```

```
 93   /**********************************************************************
 94   **
 95   ** Routine:   LockHandleMutex
 96   **
 97   ** Inputs:    None
 98   **
 99   ** Outputs:   None
100   **
101   ** Return Codes:  None
102   **
103   **
104   ** Purpose:   Lock the mutex for the handle tree object
105   **
106   **********************************************************************
107   */

109   static void
110   LockHandleMutex()
111   {
112       static boolean_ty first = TRUE;

114 1     if (first == TRUE)
115 2     {
116 2         first = FALSE;
117 2         pthread_mutex_init(&G_handleTreeMtx, NULL);
118 1     }

120 1     pthread_mutex_lock(&G_handleTreeMtx);
121   }
```

```
123   /**********************************************************************
124   **
125   ** Routine:   UnlockHandleMutex
126   **
127   ** Inputs:    None
128   **
129   ** Outputs:   None
130   **
131   ** Return Codes:  None
132   **
133   **
134   ** Purpose:   Unlock the mutex for the handle tree object
135   **
136   **********************************************************************
137   */

139   static void
140   UnlockHandleMutex()
141   {
142 1     pthread_mutex_unlock(&G_handleTreeMtx);
143   }
```

```
145   /*********************************************************************
146   **
147   ** Routine:  getStdoutSet
148   **
149   ** Inputs:   None
150   **
151   ** Outputs:  None
152   **
153   ** Return Codes:
154   **    fd_set * - the stdoutSet...
155   **
156   ** Purpose: Returns the stdoutSet fd_set after the stdoutSetModified was
157   **          copied into it. Modified is the most recent copy.
158   **
159   **
160   *********************************************************************
      */
162   int
163   getStdoutSet(fd_set *yourset, int *highhandle, int *status)
164   {
165 1    if (status == NULL)
166 2    {
167 2       return -1;
168 1    }
170 1    if (yourset == NULL || highhandle == NULL)
171 2    {
172 2       *status = HANDLEMGR_BAD_PARAM;
173 2       return -1;
174 1    }
176 1    pthread_mutex_lock(&G_fdSetMtx);
178 1    stdoutSet = stdoutSetModified;
180 1    memcpy(yourset, &stdoutSet, sizeof(fd_set));
182 1    *highhandle = highStdout;
184 1    pthread_mutex_unlock(&G_fdSetMtx);
186 1    return 0;
187 1  }
```

```
189   /*********************************************************************
190   **
191   ** Routine:  getStderrSet
192   **
193   ** Inputs:   None
194   **
195   ** Outputs:  fd_set *yourset   - the fd_set for the from descriptors
196   **           int    *highhandle - the highest handle for this set
197   **           int    *status    - there's no status to return right now
198   **                               but leave it as a placeholder
199   **
200   ** Return Codes:
201   **    int - 0 for success or -1 for failure.
202   **
203   ** Purpose: Returns the stderrSet fd_set after the stderrSetModified was
204   **          copied into it. Modified is the most recent copy.
205   **
206   **
207   *********************************************************************
      */
209   int
210   getStderrSet(fd_set *yourset, int *highhandle, int *status)
211   {
212 1    if (status == NULL)
213 2    {
214 2       return -1;
215 1    }
217 1    if (yourset == NULL || highhandle == NULL)
218 2    {
219 2       *status = HANDLEMGR_BAD_PARAM;
220 2       return -1;
221 1    }
223 1    pthread_mutex_lock(&G_fdSetMtx);
225 1    stderrSet = stderrSetModified;
227 1    memcpy(yourset, &stderrSet, sizeof(fd_set));
229 1    *highhandle = highStderr;
231 1    pthread_mutex_unlock(&G_fdSetMtx);
233 1    return 0;
234 1  }
```

```
236
237   /********************************************************************
238   **
239   ** Routine:  LookupHandleSet
240   **
241   ** Inputs:   DD_client_session_id *sess - the session ID to lookup
242   **           with
243   ** Outputs:  int *status - a place to put a status if something goes
244   **           wrong.
245   **           EDMDDHandle **hs - the handle set to return
246   **
247   ** Return Codes:
248   **           0 for success and non-zero otherwise
249   **
250   ** Purpose:  Looks up a handle set.
251   **
      ********************************************************************
252   */
253   int
254   LookupHandleSet(
255       DD_client_session_id *sess, EDMDDHandle **hs, int *status)
256   {
257       EDMDDHandle   *ret, *handleobj;

258       if (status == NULL)
259       {
260           return -1;
261       }

263       if (hs == NULL || sess == NULL)
264       {
265           *status = HANDLEMGR_BAD_PARAM;
266           return -1;
267       }

269       handleobj = new EDMDDHandle();

271       if (handleobj == NULL)
272       {
273           *status = HANDLEMGR_NO_MEMORY;
274           return -1;
275       }

277       handleobj -> setSessionID(*sess);

279       LockHandleMutex();

281       ret = (EDMDDHandle *) G_handleTree.find(handleobj);

283       UnlockHandleMutex();

285       delete handleobj;

287       if (ret == NULL)
288       {
289           *status = HANDLEMGR_LOOKUP_FAILED;
290           return -1;
291       }

293       *hs = ret;

295       return 0;
```

```
296   }
```

```
298    /***************************************************************
299    **
300    ** Routine:  newHandleSet
301    **
302    **
303    **
304    ** Inputs:   int stdouthandle - the handle to send commands to auxproc
                    int stderrhandle - the handle to receive responses from
                                       auxproc
305    **
306    **    rpc_binding_handle_t *connhandle - the connection handle
307    **    ElinkShellObjPtr_ty *shell - the shell handle
308    **
309    ** Outputs:  int *status - a place to put a status if something goes
                                  wrong.
310    **
311    ** Return Codes:
312    **     0 for success and non-zero otherwise
313    **
314    **
315    ** Purpose: Creates a new handle set.
       **
       ****************************************************************
       */

317    int
318    newHandleSet(
319        DD_client_session_id *sess, int stdouthandle, int stderrhandle,
           rpc_binding_handle_t *connhandle,
                                 ElinkShellObjPtr_ty *shell,
320        int *status)
321    {
322        EDMDDHandle *handle;
323        EDMDDHandle *ret;
324        int          flags = 0;
325        static boolean_ty first = TRUE;

327        if (first == TRUE)
328        {
329            initFdSets();
330            first = FALSE;
331        }

333        if (status == NULL)
334        {
335            return -1;
336        }

338        if (sess == NULL || connhandle == NULL || shell == NULL)
339        {
340            *status = HANDLEMGR_BAD_PARAM;
341            return -1;
342        }

344        handle = new EDMDDHandle();

346        if (handle == NULL)
347        {
348            *status = HANDLEMGR_NO_MEMORY;
349            return -1;
350        }

352        handle -> setStdoutPipe(stdouthandle);
353        handle -> setStderrPipe(stderrhandle);
354        handle -> setBindingHandle(connhandle);
355        handle -> setShellHandle(shell);
356        handle -> setSessionID(*sess);
```

```
358        pthread_mutex_lock(&G_fdSetMtx);

360        flags = fcntl(stdouthandle, F_GETFL, 0);
361        fcntl(stdouthandle, F_SETFL, flags | O_NDELAY);

363        flags = fcntl(stderrhandle, F_GETFL, 0);
364        fcntl(stderrhandle, F_SETFL, flags | O_NDELAY);

366        FD_SET(stdouthandle, &stdoutSetModified);
367        FD_SET(stderrhandle, &stderrSetModified);

369        if (stdouthandle > highStdout)
370        {
371            highStdout = stdouthandle;
372        }

374        if (stderrhandle > highStderr)
375        {
376            highStderr = stderrhandle;
377        }

379        pthread_mutex_unlock(&G_fdSetMtx);

381        LockHandleMutex();

383        ret = (EDMDDHandle *) G_handleTree.insert(handle);

385        UnlockHandleMutex();

387        if (ret == NULL)
388        {
389            *status = HANDLEMGR_INSERT_FAILED;
390            delete handle;
391            return -1;
392        }

394        return 0;
395    }
```

```
397   /************************************************************************
398   **
399   ** Routine:  getHandleSet
400   **
401   ** Inputs:   DD_client_session_id *sess - a session ID to use to look
402   **                                        up the handle set
403   **
404   ** Outputs:  int *status - a place to put a status if something goes
405   **                         wrong.
406   **           int *sout - stdout descriptor for the service.
407   **           int *serr - stderr descriptor for the service.
408   **
409   ** Return Codes:
410   **     0 for success and non-zero otherwise
411   **
412   ** Purpose:  Removes a handle set.
413   **
414   ***********************************************************************/
416   int
417   getHandleSet(
418       IN DD_client_session_id *sess, OUT int *sout, OUT int *serr,
419       OUT int *status)
420 1 {
421 1     EDMDDHandle *handle;
      1     int          lret;
423 1
424 2     if (status == NULL)
425 2     {
426 1         return -1;
          1     }
428 1
429 2     if (sess == NULL || sout == NULL || serr == NULL)
430 2     {
431 2         *status = HANDLEMGR_BAD_PARAM;
432 1         return -1;
          1     }
434 1
436 1     lret = LookupHandleSet(sess, &handle, status);
437 1
438 1     if (lret != 0)
439 1         return lret;
          1     else if (handle == NULL)
          1         return -1;
441 1
442 1     *sout = handle -> getStdoutPipe();
          1     *serr = handle -> getStderrPipe();
444 1
445 1     return 0;
          }
```

```
447   /************************************************************************
448   **
449   ** Routine:  GetCSCHandle
450   **
451   ** Inputs:   DD_client_session_id *sess - a session ID to use to look
452   **                                        up the handle set
453   **
454   ** Outputs:  int *status - a place to put a status if something goes
455   **                         wrong.
456   **           rpc_binding_handle_t *cscb - binding handle for this
457   **                                        session
458   ** Return Codes:
459   **     0 for success and non-zero otherwise
460   **
461   ** Purpose:  Returns CSC binding handle.
462   **
463   ***********************************************************************/
465   int
466   GetCSCHandle(
467       IN DD_client_session_id *sess, OUT rpc_binding_handle_t **cscb,
468       OUT int *status)
468 1 {
469 1     EDMDDHandle *handle;
470 1     int          lret;
472 1
473 2     if (sess == NULL || cscb == NULL || status == NULL)
474 2     {
475 1         return -1;
          1     }
477 1
479 1     lret = LookupHandleSet(sess, &handle, status);
480 1
481 1     if (lret != 0)
482 1         return lret;
          1     else if (handle == NULL)
          1         return -1;
484 1
486 1     *cscb = handle -> getBindingHandle();
487 1
          1     return 0;
          }
```

```
489    /****************************************************************
490    **
491    ** Routine:  GetShellHandle
492    **
493    ** Inputs:   DD_client_session_id *sess - a session ID to use to look
                     up
                     the handle set
494    **
495    **
496    ** Outputs:  int *status - a place to put a status if something goes
                     wrong.
497    **           ElinkShellObjPtr_ty **shell - shell handle for this
                     session
498    **
499    ** Return Codes:
500    **           0 for success and non-zero otherwise
501    **
502    ** Purpose:  Returns shell handle.
503    **
504    *****************************************************************
505    */
507    int
508    GetShellHandle(
509  1      IN DD_client_session_id *sess, OUT ElinkShellObjPtr_ty **shell,
               OUT int *status)
510  1    {
511  1        EDMDDHandle *handle;
512  1        int          lret;
514  1        if (sess == NULL || shell == NULL || status == NULL)
515  2        {
516  2            return -1;
517  1        }
519  1        lret = LookupHandleSet(sess, &handle, status);
521  1        if (lret != 0)
522  1            return lret;
523  1        else if (handle == NULL)
524  1            return -1;
526  1        *shell = handle -> getShellHandle();
528  1        return 0;
529  1    }
```

```
531    /****************************************************************
532    **
533    ** Routine:  deleteHandleSet
534    **
535    ** Inputs:   DD_client_session_id *sess - a session ID to use to look
                     up
                     the handle set
536    **
537    **
538    ** Outputs:  int *status - a place to put a status if something goes
                     wrong.
539    **
540    ** Return Codes:
541    **
542    **           0 for success and non-zero otherwise
543    **
544    ** Purpose:  Removes a handle set.
545    **
546    *****************************************************************
548    */
549    int
550    deleteHandleSet(DD_client_session_id *sess, ElinkHandlePtr_ty *hand,
                       int *status)
551  1    {
552  1        EDMDDHandle *handle;
553  1        EDMDDHandle *ret;
554  1        int          lret;
555  1        rpc_binding_handle_t *bh;
556  2        ElinkShellObjPtr_ty *shell;
557  1        error_status_t err;
559  1        if (status == NULL)
560  2        {
561  2            return -1;
562  1        }
564  1        if (sess == NULL)
565  2        {
566  2            *status = HANDLEMGR_BAD_PARAM;
567  2            return -1;
568  1        }
570  1        lret = LookupHandleSet(sess, &handle, status);
572  1        if (lret != 0)
573  1            return lret;
574  1        else if (handle == NULL)
575  1            return -1;
577  1        LockHandleMutex();
579  1        ret = (EDMDDHandle *) G_handleTree.remove(handle);
581  1        UnlockHandleMutex();
583  1        if (ret == NULL)
584  2        {
585  2            *status = HANDLEMGR_REMOVE_FAILED;
586  2            delete handle;
587  2            return -1;
588  1        }
590  1        pthread_mutex_lock(&G_fdSetMtx);
```

```
592  1     FD_CLR(ret -> getStdoutPipe(), &stdoutSetModified);
593  1     FD_CLR(ret -> getStderrPipe(), &stderrSetModified);

595  1     pthread_mutex_unlock(&G_fdSetMtx);

597  1     bh = ret -> getBindingHandle();

599  1     csc_free_binding(bh, 0, &err);

601  1     shell = ret -> getShellHandle();

603  1     ELinkDestroyObj(hand, *shell);

605  1     delete ret;

607  1     return 0;
608  1  }
```